REMARKS

Procedural History

Claims 1-17 are pending in the application. Claims 1-17 have been rejected. Claims 18 - 20 are new.

Response to Rejection of Claims 1 - 17

In the Office Action, the Examiner rejected Claims 1-17 under 35 U.S.C. § 103(a) as being unpatentable over Frederick, U.S. Patent No. 5,768,126 in view of Kondo, U.S. Patent No. 5,952,596 (The '596 patent"). The '596 patent was filed in the U.S. on Sept 15, 1998.

Without agreeing with the Examiner's rejection of Claims 1 - 17 under 35 U.S.C. § 103(a), Applicant submits herewith a Declaration under 35 C.F.R. § 1.131 (the "Declaration") establishing inventorship in the United States prior to the effective date of the '596 patent.

The Declaration is that of Kenneth E. Cooke, the named inventor, and includes a Declaration of Rahul Agarwal (Exhibit 1), an assistant to the inventor on the present application. The Declaration, Exhibit 1 and Appendices A, B and C attached thereto demonstrate Applicant's reduction to practice of independent Claims prior to the January 15, 1998 effective date of the Jones patent. Therefore, Applicant respectfully submits that Claims 1 - 17 should now be in condition for allowance.

The Examiner is respectfully invited to contact the undersigned should any additional information be required.

New Claims 18 - 20

Applicant has added Claims 18 - 20, which depend from Claim 1, 7 and 17. Because Claim 1, 7 & 17 and the other rejected claims should now be allowable in view of the § 1.131 Declaration, Applicant submits that Claims 18 - 20, as dependent claims, should likewise be allowable.

Summary

Applicant respectfully requests the Examiner to withdraw the rejection to Claims 1- 17 under 35 U.S.C. § 103(a), to allow Claims 18 - 20, and to pass the present application to issuance.

Should there be any remaining impediments, questions or issues, the Examiner is invited to contact Douglas G. Muehlhauser, the attorney of record at (949) 721-2994 (direct dial) or at the general office telephone number listed below.

Respectfully submitted,

KNOBBE, MARTENS, OLSON & BEAR, LLP

Dated: November 22, 2002

By: _____

Robert J. Roby
Registration No. 44,304
Attorney of Record
2040 Main Street
Fourteenth Floor
Irvine, CA 92614
(949) 760-0404

H:\DOCS\DGM\DGM-3140.DOC/bl
112202

Appendix A

```
/*****************************************************************
**********
 *
 *
 *
 *  Copyright (C)  Progressive Networks.
 *  All rights reserved.
 *
 * Fixed-point crossfade.
 * Uses arbitrary lookup table, with table interpolation.
 *
 * Ken Cooke
 */

#include <stdio.h>
#include <math.h>
#include "pnresult.h"
#include "pntypes.h"
#include "pnassert.h"

#include "crosfade.h"

#include "pnheap.h"
#ifdef _DEBUG
#undef PN_THIS_FILE
static char PN_THIS_FILE[] = __FILE__;
#endif

//#define GENERATE_TABLE        1
#define TABLE_FILE_NAME     "table.cpp"


CrossFader::CrossFader()
    : tabstep(0)
    , tabacc(0)
    , tabint(0)
    , m_uNumChannels(0)
    , m_bInitialized(FALSE)
{
}

CrossFader::~CrossFader()
{
}

PN_RESULT
```

```
CrossFader::Initialize(UINT16 uNumSamplesToFadeOn, UINT16 uNumChann
els)
{
    PN_ASSERT(uNumSamplesToFadeOn > 0 && uNumChannels > 0);

    if (uNumSamplesToFadeOn == 0 || uNumChannels == 0)
    {
        return PNR_INVALID_PARAMETER;
    }

    tabacc = 0;          /* accumulator */
    tabint = 0;          /* table index */
    tabstep = (NALPHA << FRACBITS) / uNumSamplesToFadeOn;

    m_uNumChannels = uNumChannels;
    m_bInitialized  = TRUE;

    return PNR_OK;
}

/*
 * Crossfades over nfade samples, operating in-place on sampnew.
 * alpha values are generated from table via linear interpolation.
 */
void
CrossFader::CrossFade(INT16* sampold, INT16* sampnew, UINT16 uNumSa
mples)
{
    INT32 alerp, sdelta;
    UINT16 n;

    if (!m_bInitialized)
    {
        return;
    }

    /* crossfade, stepping thru table in fixed point */
    for (n = 0; n < uNumSamples; n++)
    {
        /* interpolate new alpha */
        alerp = alpha[tabint];
        alerp += (adelta[tabint] * (tabacc & FRACMASK)) >> FRACBITS
;
        /* next table step */
        tabacc += tabstep;
        tabint = tabacc >> FRACBITS;

        for (UINT16 uNumChannels = 0; uNumChannels < m_uNumChannels
```

```
; uNumChannels++)
        {
            /* crossfade, using interpolated alpha */
            sdelta = alerp * (*sampold - *sampnew);
            *sampnew += int((sdelta + FRACROUND) >> FRACBITS);

            sampold++;
            sampnew++;
        }
    }
}


/*
 * Initialize crossfade tables.
 * Currently using linear dB, but could be anything...
 * This code is ONLY used to generate static tables.
 */
void
CrossFader::CrossFadeInit(void)
{
    double dbval, dbstep;
    int n;

#ifdef GENERATE_TABLE
    FILE* fd = fopen(TABLE_FILE_NAME, "w+");
    fprintf(fd, "static INT32 alpha[NALPHA+1] = {\n");
    fprintf(fd, "\n\t\t\t\t");
#endif /*GENERATE_TABLE*/

    /* generate alpha table */
    dbstep = (DBEND - DBSTART) / (NALPHA - 1);
    for (n = 0; n < NALPHA; n++)
    {
        /* linear steps in dB */
        dbval = DBSTART + (dbstep * n);
        /* store as gains, in fixed point */
        alpha[n] = (int) (DB2GAIN(dbval) * (1<<FRACBITS) + 0.5);

#ifdef GENERATE_TABLE
        fprintf(fd, "%d, ", alpha[n]);
        if ((n+1) % 5 == 0)
        {
            fprintf(fd, "\n\t\t\t\t");
        }
#endif /*GENERATE_TABLE*/
    }
```

```
    alpha[n] = alpha[n-1];      /* in case roundoff oversteps */

#ifdef GENERATE_TABLE
    fprintf(fd, "%d\n", alpha[n]);
    fprintf(fd, "};\n\n");

    fprintf(fd, "static INT32 adelta[NALPHA] = {\n");
    fprintf(fd, "\n\t\t\t\t");
#endif /*GENERATE_TABLE*/

    /* generate delta table, for fast interpolate */
    for (n = 0; n < NALPHA-1; n++)
    {
        adelta[n] = alpha[n+1] - alpha[n];

#ifdef GENERATE_TABLE
        fprintf(fd, "%d, ", adelta[n]);
        if ((n+1)%5 == 0)
        {
            fprintf(fd, "\n\t\t\t\t");
        }
#endif /*GENERATE_TABLE*/
    }

    adelta[n] = 0;      /* in case roundoff oversteps */

#ifdef GENERATE_TABLE
    fprintf(fd, "%d\n", adelta[n]);
    fprintf(fd, "};\n\n");
    fclose(fd);
#endif /*GENERATE_TABLE*/
}

/*
 * test app
 */

#ifdef GENERATE_TABLE

#define NMAX 4096
short testold[NMAX];
short testnew[NMAX];
int
main(void)
{
        int n;

        for (n = 0; n < NMAX; n++) {
```

```
                testold[n] = -32768;
                testnew[n] = 32767;
        }

        CrossFadeInit();
        CrossFade(testold, testnew, 1024);

        return 0;
}
#endif /*GENERATE_TABLE*/
```

Appendix B

```
/*****************************************************************
**********
 *  pnaustr.cpp
 *
 *  Copyright (C)  Progressive Networks.
 *  All rights reserved.
 *
 *  Progressive Networks Confidential and Proprietary information.
 *  Do not redistribute.
 *
 *  PN implementation of Audio Stream Interface.
 *
 *
 */

#include <stdio.h>
#include <string.h>



#include "pnresult.h"
#include "pntypes.h"

#include "pncom.h"
#include "rmaengin.h"
#include "rmapckts.h"
#include "rmaausvc.h"
#include "rmarasyn.h"

#include "pnpckts.h"
#include "pnaudstr.h"
#include "pnaudply.h"
#include "pnaudses.h"
#include "pnmixer.h"
#include "pnaudvol.h"   // for IRMAVolume

#include "pnslist.h"
#include "pnmap.h"
#include "auderrs.h"

#include "pntick.h"

#include "resample.h"   // for resampler
#include "interp.h"     // for Linear Interpolator
#include "crosfade.h"
```

```
...

/*************************************************************************
******
 *  Method:
 *      CPNAudioStream::MixIntoBuffer
 *  Purpose:
 *      Mix stream data into this pPlayerBuf.
 *  Note:
 *
 *      Resampler always works on 16 bit PCM. If the input is
 *      8 bit, it converts it first to 16 bit before making
 *      any resampling calculations.
 *      We always try to open audio device in 16 bit stereo mode.
 *      This is because a new player/stream may be instantiated in
the
 *      midst of a presentation and this new stream may be stereo.
If we
 *      earlier opened the device as mono, we will have to force th
is stereo
 *      stream to be played as mono! Not a good idea. Also any mono
-streo
 *      conversion almost comes for free (some extra memory usage a
nd
 *      an extra assignment) since it can be done in the mixing loo
p in
 *      MixBuffer().
 *
 *      Any 8-16 and stereo-mono conversion, if required, SHOULD be
 done
 *      before resampling.
 *      Any mono-stereo conversion should be done after resampling.
 *
 *      Stereo-Mono conversion code resides in the resampler.
 *      Mono-Stereo conversion code resides in the mixer.
 *
 */
PN_RESULT CPNAudioStream::MixIntoBuffer
(
    UCHAR*   pPlayerBuf,
    ULONG32  ulBufSize,
    ULONG32& ulBufTime,
    BOOL     bCalledFromAudioThread,
    BOOL     bGetCrossFadeData
)
{
    BOOL    bCrossFadeThisTime   = FALSE;
    UINT32  ulTimeActuallyFaded = m_ulGranularity;
```

```
    if (!m_bInited)
    {
        return PNR_NOT_INITIALIZED;
    }

//{FILE* f1 = ::fopen("c:\\temp\\rasync.txt", "a+"); ::fprintf(f1,
"Call MixIntoBuffer: %lu\n", m_ulLastWriteTime);::fclose(f1);}
    /* If this is a *FROM* stream, we may have already mixed
     * data during cross-fade with *TO* stream
     */
    if (m_bFadeAlreadyDone && !m_bFadeToThisStream)
    {
        m_bFadeAlreadyDone = FALSE;
        return PNR_OK;
    }

    PN_ASSERT(!bGetCrossFadeData || !m_bFadeToThisStream);

    /* If we need to mix cross fade data from the *from* stream,
     * it better be available
     */
    PN_ASSERT(!bGetCrossFadeData || m_pDataList->GetCount() > 0);

    /* If this stream needs to be cross-faded and is a
     * NOT a fade-to stream, it would have been already taken
     * care of by the fade-to stream in an earlier call to
     * MixIntoBuffer()
     */
    if (m_bCrossFadingToBeDone && m_pDataList->GetCount() > 0)
    {
        PNAudioInfo* pInfo = (PNAudioInfo*) m_pDataList->GetHead();
        UINT32 ulStartTime  = 0;
        if (pInfo)
        {
            ulStartTime = pInfo->ulStartTime +
                          CalcMs(pInfo->pBuffer->GetSize() -
                                 pInfo->ulBytesLeft);
            if (m_bFadeToThisStream)
            {
                /* Cool! It is time for cross-fading */
                if ((m_ulLastWriteTime <= m_ulCrossFadeStartTime &&
                    m_ulCrossFadeStartTime - m_ulLastWriteTime <= m
_ulGranularity) ||
                    m_ulLastWriteTime > m_ulCrossFadeStartTime &&
                    m_ulLastWriteTime - m_ulCrossFadeStartTime <= m
_ulFudge)
                {
                    bCrossFadeThisTime = TRUE;
```

```
                    if (m_ulLastWriteTime <= m_ulCrossFadeStartTime
)
                    {
                        ulTimeActuallyFaded = m_ulGranularity -
                            (m_ulCrossFadeStartTime - m_ulLastWrite
Time);
                    }

//{FILE* f1 = ::fopen("c:\\raroot\\racross.txt", "a+"); ::fprintf(f
1, "m_ulLastWriteTime: %lu ulStartTime: %lu m_ulCrossFadeStartTime:
 %lu pInfo->ulStartTime: %lu pInfo->pBuffer->GetSize(): %lu pInfo->
ulBytesLeft: %lu\n", m_ulLastWriteTime, ulStartTime, m_ulCrossFadeS
tartTime, pInfo->ulStartTime, pInfo->pBuffer->GetSize(), pInfo->ulB
ytesLeft);::fclose(f1);}
                    PN_ASSERT(
                        (ulStartTime >= m_ulCrossFadeStartTime &&
                        (ulStartTime <= m_ulCrossFadeStartTime +
                                        m_ulCrossFadeDuration)) ||
                        (ulStartTime < m_ulCrossFadeStartTime &&
                        (m_ulCrossFadeStartTime - ulStartTime <= m_
ulFudge)));
                }
            }
            /* We better have enough data available to cross-fade *
/
            else if (bGetCrossFadeData)
            {
#if 0
                RemoveExcessCrossFadeData();
                if (m_pDataList->GetCount() > 0)
                {
                    pInfo = (PNAudioInfo*) m_pDataList->GetHead();
                    ulStartTime = pInfo->ulStartTime +
                        CalcMs(pInfo->pBuffer->GetSize() - pInfo->u
lBytesLeft);
                    PN_ASSERT(ulStartTime >= m_ulCrossFadeStartTime
 &&
                        ulStartTime <= m_ulCrossFadeStartTime +
                                       m_ulCrossFadeDuration);
                }
                else
                {
                    /* We should have data here for cross-fading*/
                    PN_ASSERT(FALSE);
                }
#endif
            }
        }
```

```
    }

    if (!bGetCrossFadeData && ulBufTime < m_ulLastWriteTime)
    {
        ulBufTime = m_ulLastWriteTime;
    }

    /* If there are any DryNotifications and the data list is empty
     * we need to notify them so that they can write more data.
     */
    if (m_DryNotificationMap.GetCount() > 0)
    {
        ULONG32 ulNumMsRequired = m_ulGranularity;
        ULONG32 ulLastWriteTime = m_ulLastWriteTime;
        if (m_pDataList->IsEmpty() || !EnoughDataAvailable(ulLastWr
iteTime, ulNumMsRequired))
        {
            if (bCalledFromAudioThread)
            {
                return PNR_FAIL; //PNR_NOT_ENOUGH_DATA;
            }

            IRMADryNotification* pDryNotification = 0;
            CPNMapPtrToPtr::Iterator lIter = m_DryNotificationMap.B
egin();
            for (; lIter != m_DryNotificationMap.End(); ++lIter)
            {
                pDryNotification = (IRMADryNotification*) (*lIter);
                pDryNotification->OnDryNotification(ulLastWriteTime
, ulNumMsRequired);
            }

            if (m_Owner->GetState() != E_PLAYING)
            {
                return PNR_OK;
            }
        }
    }

    // ////////////////////////////////////////////////////////////
/
    // There may be no buffers in the list. No packets? Play silenc
e.
    // Still need to increment time.
    if ( m_pDataList->IsEmpty() && m_pInstantaneousList->IsEmpty()
)
    {
        m_ulLastWriteTime += m_ulGranularity;
```

```
        return PNR_NO_DATA;
    }

    UCHAR*      pSourceBuffer = 0;
    ULONG32     ulMaxBytes = 0;
    ULONG32     ulMaxFramesIn = 0;
    ULONG32     ulMaxFramesOut = 0;
    ULONG32     ulNumBytesMixed = 0;
    BOOL        bMonoToStereoMayBeConverted = TRUE;


    // ////////////////////////////////////////////////////////////
/
    // If no  resampling, mix stream data directly into the player
    // buffer.
    if ( !m_pResampleId && !m_pInterpId)
    {
        pSourceBuffer   = pPlayerBuf;
        ulMaxBytes      = m_ulInputBytesPerGran;

        /* For only those sound cards which do not
         * support stereo - a RARE (non-existent) case
         */
        if (m_AudioFmt.uChannels == 2 && m_DeviceFmt.uChannels == 1
)
        {
            /* We should never reach here since this case
             * should be handled by the Resampler
             * Temporary ASSERT...
             */
            PN_ASSERT(FALSE);
        }
        /* Mono->Stereo conversion*/
        else if (m_bChannelConvert)
        {
            PN_ASSERT(ulMaxBytes <= ulBufSize/2);

            /* Avoid GPF in retail builds! */
            if (ulMaxBytes > ulBufSize/2)
            {
                ulMaxBytes = ulBufSize/2;
            }
        }
        else
        {
            PN_ASSERT(ulMaxBytes <= ulBufSize);

            /* Avoid GPF in retail builds! */
```

```
            if (ulMaxBytes > ulBufSize)
            {
                ulMaxBytes = ulBufSize;
            }
        }
    }
    // ////////////////////////////////////////////////////////////
/
    // If resampling, mix stream data into a tmp buffer. Then
    // resample this buffer and mix the final resampled buffer into
    // the player buffer.
    else
    {
        bMonoToStereoMayBeConverted = FALSE;

        memset(m_pTmpResBuf, 0, PN_SAFESIZE_T(m_ulMaxBlockSize));
        pSourceBuffer = m_pTmpResBuf;

        /*
         * Audio Session will always ask for m_ulOutputBytesPerGran
 bytes to be mixed
         * in MixIntoBuffer() call. So we need to produce these man
y number of bytes.
         * If there is any mono-stereo conversion that happens in t
he mixing, number
         * of output bytes required from the resampler are half the
 number of
         * m_ulOutputBytesPerGran bytes.
         */

        if (m_pResampleId)
        {
            ulMaxFramesOut = m_ulOutputBytesPerGran/(m_DeviceFmt.uB
itsPerSample==8 ? 1 : 2);

            if (m_DeviceFmt.uChannels == 2)
            {
                ulMaxFramesOut /= 2;
            }

            ulMaxFramesIn = ResamplerRequires(ulMaxFramesOut, m_pRe
sampleId);

            ulMaxBytes = ulMaxFramesIn *  ((m_AudioFmt.uBitsPerSamp
le==8)? 1 : 2)
                                          * m_AudioFmt.uChannels;
        }
#ifdef LINEAR_INTER
```

```
        else /* if (m_pInterpId) */
        {
            ulMaxBytes = m_ulInputBytesPerGran;
        }
#endif

        PN_ASSERT(ulMaxBytes <= m_ulMaxBlockSize);
    }

    // ////////////////////////////////////////////////////////////////
/
    // Mix n bytes of data into buffer
    ulNumBytesMixed = MixData(pSourceBuffer, ulMaxBytes, bMonoToSte
reoMayBeConverted);

/*
    if (ulNumBytesMixed > 0)
    {
        ::fwrite(pSourceBuffer, ulNumBytesMixed, 1, fdbefore);
    }
*/

    // ////////////////////////////////////////////////////////////////
/
    // If we need to resample , then do this and then mix data into
    // the player buffer.
    // Only resample and mix if volume is *not* zero and there
    // are some bytes to mix.
    if ((m_pResampleId || m_pInterpId)
        && ulNumBytesMixed > 0 && m_uVolume > 0 && !m_bMute)
    {
        if(ulNumBytesMixed < ulMaxBytes  &&  8==m_AudioFmt.uBitsPer
Sample)
        {
            //fill remainder with 128's (-1), silence:
            UCHAR* pTmp = &pSourceBuffer[ulNumBytesMixed];
            ULONG32 ulNumBytesLeftToSilence = ulMaxBytes-ulNumBytes
Mixed;
            do
            {
                *pTmp = 128;
                pTmp++;
            } while(--ulNumBytesLeftToSilence);
        }
        BOOL bChannelConvert    = FALSE;
        ULONG32 ulOutBytes      = 0;
        if (m_pResampleId)
        {
```

```
            ulOutBytes = Resample( (short*)pSourceBuffer,
                                   ulMaxBytes,
                                   (short*)m_pResampleBuf,
                                   m_pResampleId,
                                   &bChannelConvert);

//          ::fwrite(m_pResampleBuf, ulOutBytes, 1, fdafter);
            /* Resampler will do stereo to mono conversion for us.*
/
            PN_ASSERT(ulMaxFramesOut == (ulOutBytes / 2 /
                (m_AudioFmt.uChannels == 2 && m_DeviceFmt.uChannels
 == 1 ? 1 : m_AudioFmt.uChannels))) ;

            PN_ASSERT(m_bChannelConvert == bChannelConvert);
        }
#ifdef LINEAR_INTER
        else /* if (m_pInterpId) */
        {
            UINT32 ulSampleSize = m_AudioFmt.uChannels *
                                  (m_AudioFmt.uBitsPerSample == 16
? 2 : 1);
            //Interp(short *inbuf, long insamps, short *outbuf, voi
d *id)
            UINT32 ulOutSamples = Interp( (short*)pSourceBuffer,
                                    ulMaxBytes/ulSampleSize,
                                    (short*)m_pResampleBuf,
                                    m_pInterpId);

            if (m_AudioFmt.uChannels == 2 && m_DeviceFmt.uChannels
== 1)
            {
                ulOutBytes = ulOutSamples *
                         (m_DeviceFmt.uBitsPerSample == 16 ? 2 : 1)
;
            }
            else
            {
                ulOutBytes = ulOutSamples * m_AudioFmt.uChannels *
                         (m_DeviceFmt.uBitsPerSample == 16 ? 2 : 1)
;
            }
        }
#endif

        if (m_bChannelConvert)
        {
            PN_ASSERT(m_pInterpId || ulOutBytes*2 <= ulBufSize);
            if ( ulOutBytes > ulBufSize/2 )
```

```
                ulOutBytes = ulBufSize/2;
        }
        else
        {
            PN_ASSERT(m_pInterpId || ulOutBytes <= ulBufSize);
            if ( ulOutBytes > ulBufSize )
                ulOutBytes = ulBufSize;
        }

        CPNMixer::MixBuffer( m_pResampleBuf, pPlayerBuf,
                             ulOutBytes, m_bChannelConvert,
                             m_uVolume, m_DeviceFmt.uBitsPerSample)
;
    }

#ifdef _TESTING
    if ( g_log > 0 )
    {
        write( g_log, pPlayerBuf, ulNumBytesMixed);
    }

#endif

    /* This is for *FROM* stream */
    if (bGetCrossFadeData)
    {
        m_bFadeAlreadyDone = TRUE;
    }
    /* If we are cross-fading, we have data from this stream in pPl
ayerBuf
     * Now get data to be  cross-faded from *From* stream in
     * m_pCrossFadeBuffer
     */
    else if (bCrossFadeThisTime)
    {
        /* Allocate CrossFade Buffer */
        if (!m_pCrossFadeBuffer)
        {
            m_ulCrossFadeBufferSize = ulBufSize;
            m_pCrossFadeBuffer = new UCHAR[m_ulCrossFadeBufferSize]
;
        }

        memset(m_pCrossFadeBuffer, 0, PN_SAFESIZE_T(m_ulCrossFadeBu
fferSize));

        UINT32 ulCrossFadeLen = m_ulCrossFadeBufferSize;
        UINT32 ulTmpBufTime   = 0;
```

```
        m_pCrossFadeStream->MixIntoBuffer(m_pCrossFadeBuffer,
                                          ulCrossFadeLen, ulTmpBufTi
me, bCalledFromAudioThread, TRUE);
        /* Now it is time to perform cross-fading between
         * pPlayerBuf and m_pCrossFadeBuffer
         */

        UINT32 ulStartByteToFade            = 0;
        UINT32 ulNumMsInThisBuffer          = CalcDeviceMs(ulBufSiz
e);
        UINT32 ulNumBytesToBeCrossFaded = ulBufSize;
        UINT32 ulSampleSize = ((m_DeviceFmt.uBitsPerSample==8)? 1 :
 2)
                            * m_DeviceFmt.uChannels;

        /* Make sure we have integral number of samples */
        PN_ASSERT(ulBufSize == (ulBufSize/ulSampleSize) * ulSampleS
ize);


        /* Only partial buffer needs to be cross-faded.
         *   -------------
         * ~~_____________|
         *
         *       -----------
         *      |___________~~
         *
         *
         *     -----
         *    |_____|  <-- Granularity size block that is mixed.
         *
         *       <--> Only partial block needs to be faded
         */

        if (ulTimeActuallyFaded < ulNumMsInThisBuffer)
        {
            ulNumBytesToBeCrossFaded = (UINT32) (ulBufSize *
                (ulTimeActuallyFaded*1./ulNumMsInThisBuffer)) ;

            UINT32 ulOutOfPhase = ulNumBytesToBeCrossFaded % ulSamp
leSize;
            if (ulOutOfPhase > 0)
            {
                ulNumBytesToBeCrossFaded =
                    ulNumBytesToBeCrossFaded - ulOutOfPhase;
            }

            ulStartByteToFade = ulBufSize - ulNumBytesToBeCrossFade
```

```
d;
        }

        if (ulTimeActuallyFaded > m_ulCrossFadeDuration)
        {
            ulNumBytesToBeCrossFaded = (UINT32) (ulBufSize *
                (m_ulCrossFadeDuration*1./ulNumMsInThisBuffer)) ;

            UINT32 ulOutOfPhase = ulNumBytesToBeCrossFaded % ulSamp
leSize;
            if (ulOutOfPhase > 0)
            {
                ulNumBytesToBeCrossFaded =
                    ulNumBytesToBeCrossFaded - ulOutOfPhase;
            }
        }

        UINT16 uNumSamples = (UINT16) (ulNumBytesToBeCrossFaded/
                                       ulSampleSize);
        m_pCrossFader->CrossFade((INT16*) (m_pCrossFadeBuffer+ulSta
rtByteToFade),
                                 (INT16*) (pPlayerBuf+ulStartByteTo
Fade),
                                 uNumSamples);

        /* Mix the initial bytes that are not cross-faded*/
        if (ulStartByteToFade > 0)
        {
            CPNMixer::MixBuffer(m_pCrossFadeBuffer,
                                pPlayerBuf,
                                ulStartByteToFade,
                                FALSE,
                                100,
                                m_DeviceFmt.uBitsPerSample);
        }

        /* Mix the remaining bytes */
        if (ulNumBytesToBeCrossFaded + ulStartByteToFade < ulBufSiz
e)
        {
            CPNMixer::MixBuffer(m_pCrossFadeBuffer + ulNumBytesToBe
CrossFaded + ulStartByteToFade,
                                pPlayerBuf + ulNumBytesToBeCrossFad
ed + ulStartByteToFade,
                                ulBufSize - (ulNumBytesToBeCrossFad
ed + ulStartByteToFade),
                                FALSE,
                                100,
```

```
                                    m_DeviceFmt.uBitsPerSample);
        }
    }

    if (bGetCrossFadeData || bCrossFadeThisTime)
    {
        if (bGetCrossFadeData)
        {
            PN_ASSERT(m_ulLastWriteTime >= m_ulCrossFadeStartTime);
            if (m_ulLastWriteTime >= m_ulCrossFadeStartTime)
            {
                ulTimeActuallyFaded = m_ulLastWriteTime - m_ulCross
FadeStartTime;
            }
        }

        if (ulTimeActuallyFaded >= m_ulCrossFadeDuration)
        {
            m_bCrossFadingToBeDone = FALSE;
            PN_RELEASE(m_pCrossFadeStream);

            /* We should release any extra buffers if it is a
             * *from* stream
             */
            if (!m_bFadeToThisStream)
            {
                /* Do not remove any instantaenous buffers */
                FlushBuffers(FALSE);
            }
        }
        else
        {
            m_ulCrossFadeDuration   -= ulTimeActuallyFaded;
            m_ulCrossFadeStartTime  += ulTimeActuallyFaded;
        }
    }
    else if (m_bCrossFadingToBeDone && !m_bFadeToThisStream)
    {
        m_pCrossFadeStream->SyncStream(m_ulLastWriteTime);
    }

    return PNR_OK;
}


/*************************************************************
******
 *  Method:
```

```
 *      CPNAudioStream::MixData
 *  Purpose:
 *      Mix all valid data in my auxilliary list into the buffer.
 *
 *  Thoughts:
 *      while there are buffers available
 *          if (buffertime is more than endtime) break;
 *          if (any part of buffer is >startime and < endtime)
 *              we are in business.
 *              mix that part of the buffer, update offset,
 *              update max of num bytes written in this round.
     * Looks like we need to keep LastWrite time and offsets for al
l buffers
     * that get written in one pass.
     * Consider this scenario:
     *
     *
        ----------------- -----------------
       |_________________| |_________________|  -> Skew  1

         ________ __________
        |________| |__________|                ->Skew    2  5

         ____________________________________
        |____________________|_|_______________| Skew in opposite di
rection 3

              ________
             |________|                          4


         _____________
        |_____________|      <- Buffer to be mixed currently


          Order of buffer processing will be in the order of number
s on the right.
          Since all the theree buffers have fudge within fudge limi
t, they need to
          be written one after the other. This is possible only if
we keep last
          write times and last written offsets in mixer buffer for
each one of them.

        Is this extra processsing on every write worthed OR do we p
lace limitations
        on the data that can be written.

     *  Hmmm... We are now going with STREAMED/INSTANTANEOUS/TIMED
model since the
     *  above case is shows that users can really screw things up a
nd it would be
```

```
     *  very difficult to handle this case. So instead, we do not s
upport over-lapped
     *  buffers any more. i.e. If a renderer wants to have streamed
 and instantaneous
     *  behavior, it needs to use two audio streams.

 */
ULONG32 CPNAudioStream::MixData
(
        UCHAR*  pDestBuf
,       ULONG32 ulBufLen
,       BOOL    bMonoToStereoMayBeConverted
)
{
    PNAudioInfo*    pInfo                       = 0;
    ULONG32         ulNumBytes                  = 0;
    LISTPOSITION    lp                          = 0;
    LISTPOSITION    lastlp                      = 0;
    ULONG32         ulNumBytesWritten           = 0;
    BOOL            bLastWriteTimeToBeUpdated   = TRUE;

    // ////////////////////////////////////////////////////////////
/
    // Go thru the buffer list and mix in valid buffers.

    /* First all instantaneous buffers
     * All the instantaneous buffers get written at the start of
     * the destination buffer.
     */
    lp = lastlp = 0;
    lp = m_pInstantaneousList->GetHeadPosition();

    while( lp )
    {
        lastlp  = lp;
        pInfo   = (PNAudioInfo*) m_pInstantaneousList->GetNext(lp);

        ulNumBytes = pInfo->ulBytesLeft > ulBufLen ?
                     ulBufLen : pInfo->ulBytesLeft;

        // ////////////////////////////////////////////////////////
/////
        // Mix the data into the stream buffer, with stream volume.
 Don't
        // do volume calculation if volume is 1. If volume is 0, th
en
        // don't mix. Don't mix if there are no bytes to mix.
        if  ( m_uVolume > 1 &&  ulNumBytes > 0 )
```

```
        {
            CPNMixer::MixBuffer( pInfo->pOffset, pDestBuf,
                                 ulNumBytes, m_bChannelConvert && b
MonoToStereoMayBeConverted,
                                 m_uVolume, m_AudioFmt.uBitsPerSamp
le );
        }

        ULONG32 ulActualBytesWritten = ((m_bChannelConvert && bMono
ToStereoMayBeConverted) ?
                                        2*ulNumBytes : ulNumBytes);

        if (ulNumBytesWritten < ulActualBytesWritten)
        {
            ulNumBytesWritten = ulActualBytesWritten;
        }

        pInfo->ulBytesLeft  -= ulNumBytes;
        pInfo->pOffset      += ulNumBytes;

        if (pInfo->ulBytesLeft == 0)
        {
            pInfo->pBuffer->Release();
            delete pInfo;
            m_pInstantaneousList->RemoveAt(lastlp);
        }
    }

    /* now timed buffers */
    /* We do not support over-lapped buffers any more */
    ULONG32 ulDestinationOffset = 0;
    UINT32  ulNumInBytesWritten = 0;
    lp = lastlp = 0;
    lp = m_pDataList->GetHeadPosition();
    while( lp )
    {
        lastlp  = lp;
        pInfo   = (PNAudioInfo*) m_pDataList->GetNext(lp);

        /* only at the start of the buffer, we check whether
         * we are done or not for this round
         */
        if (pInfo->uAudioStreamType == STREAMING_AUDIO  &&
            pInfo->pOffset == pInfo->pBuffer->GetBuffer() &&
            pInfo->ulStartTime > m_ulLastWriteTime &&
            pInfo->ulStartTime - m_ulLastWriteTime > m_ulFudge)
        {
            break;
```

```
        }

        if (pInfo->uAudioStreamType == TIMED_AUDIO &&
            (pInfo->ulBytesLeft == pInfo->pBuffer->GetSize() ||
             m_bCrossFadingToBeDone))
        {
            UINT32 ulStartTime = pInfo->ulStartTime +
                                 CalcMs(pInfo->pBuffer->GetSize() -
                                    pInfo->ulBytesLeft);

            if (ulStartTime >= m_ulLastWriteTime &&
                ulStartTime - m_ulLastWriteTime >= m_ulGranularity)
            {
                break;
            }

            /*Calculate the actual time where to write from*/
            ULONG32 ulBytesDiff = 0;
            if (ulStartTime >= m_ulLastWriteTime)
            {
                ulBytesDiff = CalcOffset(m_ulLastWriteTime, ulStart
Time);
                ulBytesDiff = ulBytesDiff - (ulBytesDiff % (2*m_Aud
ioFmt.uChannels));

                if (m_bChannelConvert && bMonoToStereoMayBeConverte
d)
                {
                    ulBytesDiff *= 2;

                    /* This check is needed to account for lost pac
kets.
                     * If there are more than one packet missing in
 a row,
                     * we may get multiple packets with TIMED_AUDIO
 flag
                     * and they may be far apart in time to be writ
ten
                     * in this block
                     */
                    if (ulDestinationOffset + ulBytesDiff > ulBufLe
n*2)
                    {
                        bLastWriteTimeToBeUpdated = FALSE;
                        m_ulLastWriteTime +=
                            CalcMs((ulBufLen*2 - ulDestinationOffse
t)/2);
                        break;
```

```
                    }
                }
                else
                {
                    /* This check is needed to account for lost pac
kets.
                     * If there are more than one packet missing in
 a row,
                     * we may get multiple packets with TIMED_AUDIO
 flag
                     * and they may be far apart in time to be writ
ten
                     * in this block
                     */
                    if (ulDestinationOffset + ulBytesDiff > ulBufLe
n)
                    {
                        bLastWriteTimeToBeUpdated = FALSE;
                        m_ulLastWriteTime +=
                            CalcMs(ulBufLen - ulDestinationOffset);
                        break;
                    }
                }

                ulDestinationOffset += ulBytesDiff;
            }
            else
            {
                ulBytesDiff = CalcOffset(ulStartTime, m_ulLastWrite
Time);
                /* Make sure that it is at even byte boundary */
                ulBytesDiff     = ulBytesDiff - (ulBytesDiff % (2*m
_AudioFmt.uChannels));

                if (pInfo->ulBytesLeft >= ulBytesDiff)
                {
                    pInfo->pOffset          += ulBytesDiff;
                    pInfo->ulBytesLeft  -= ulBytesDiff;
                }
                else
                {
                    pInfo->pOffset          += pInfo->ulBytesLeft;
                    pInfo->ulBytesLeft   = 0;
                }
            }
        }

        /* just place it at the end of the last write position */
```

```
        ULONG32 ulNumMoreBytesToWrite = 0;

        if (m_bChannelConvert && bMonoToStereoMayBeConverted)
        {
            ulNumMoreBytesToWrite = (ulBufLen*2-ulDestinationOffset
)/2;
        }
        else
        {
            ulNumMoreBytesToWrite = ulBufLen-ulDestinationOffset;
        }

        ulNumBytes = pInfo->ulBytesLeft > ulNumMoreBytesToWrite ?
                     ulNumMoreBytesToWrite : pInfo->ulBytesLeft;

        // ////////////////////////////////////////////////////////
/////
        // Mix the data into the stream buffer, with stream volume.
 Don't
        // do volume calculation if volume is 1. If volume is 0, th
en
        // don't mix. Don't mix if there are no bytes to mix.
        if  (m_uVolume > 1 &&  ulNumBytes > 0)
        {
            if (m_bChannelConvert && bMonoToStereoMayBeConverted)
            {
                PN_ASSERT(ulDestinationOffset + ulNumBytes*2 <= ulB
ufLen*2);
            }
            else
            {
                PN_ASSERT(ulDestinationOffset + ulNumBytes <= ulBuf
Len);
            }

            CPNMixer::MixBuffer(pInfo->pOffset, pDestBuf+ulDestinat
ionOffset,
                                ulNumBytes, m_bChannelConvert && b
MonoToStereoMayBeConverted,
                                m_uVolume, m_AudioFmt.uBitsPerSamp
le);
        }

        ULONG32 ulActuallyBytesWritten = ulNumBytes;
        ulActuallyBytesWritten = ((m_bChannelConvert && bMonoToSter
eoMayBeConverted) ?
            ulActuallyBytesWritten*2 : ulActuallyBytesWritten);
```

```
        if (ulNumBytesWritten < ulActuallyBytesWritten + ulDestinat
ionOffset)
        {
            ulNumBytesWritten = ulActuallyBytesWritten + ulDestinat
ionOffset;
        }

        ulDestinationOffset     += ulActuallyBytesWritten;

        pInfo->ulBytesLeft  -= ulNumBytes;
        pInfo->pOffset      += ulNumBytes;

        ulNumInBytesWritten += ulNumBytes;

        ULONG32 ulLastWriteTime = pInfo->ulStartTime +
                        CalcMs(pInfo->pBuffer->GetSize() - pInfo->u
lBytesLeft);

        if (bLastWriteTimeToBeUpdated)
        {
            m_ulLastStartTimePlayed = ulLastWriteTime - CalcMs(ulNu
mBytes);
        }

        bLastWriteTimeToBeUpdated = FALSE;
        if (m_ulLastWriteTime < ulLastWriteTime)
        {
            m_ulLastWriteTime = ulLastWriteTime;
        }

        if (pInfo->ulBytesLeft == 0)
        {
            pInfo->pBuffer->Release();
            delete pInfo;
            m_pDataList->RemoveAt(lastlp);
        }

        /* Have we written enough for this time? */
        if (ulBufLen == ulDestinationOffset ||
            (m_bChannelConvert && bMonoToStereoMayBeConverted &&
            (ulBufLen*2 == ulDestinationOffset)))
        {
            break;
        }
    }

    if (bLastWriteTimeToBeUpdated)
    {
```

```
        m_ulLastWriteTime += m_ulGranularity;
    }
    else /* We mixed some input bytes */
    {
        m_ulLastEndTimePlayed = m_ulLastStartTimePlayed +
                                CalcMs(ulNumInBytesWritten);
        if (m_bRealAudioStream)
        {
//{FILE* f1 = ::fopen("c:\\temp\\rasync.txt", "a+"); ::fprintf(f1,
"Call MapFudgedTimestamps: %lu\n", m_ulLastStartTimePlayed);::fclos
e(f1);}
            MapFudgedTimestamps();
        }
    }

    return ulNumBytesWritten;
}

/*****************************************************************
******
 *  Method:
 *              CPNAudioStream::CalcMs
 *      Purpose:
 *              Calculate the duration in millisecs for this number
 of bytes.
 */
ULONG32 CPNAudioStream::CalcMs
(
    ULONG32     ulNumBytes
)
{
    return ( (ULONG32) (( 1000.0
                / (m_AudioFmt.uChannels * ((m_AudioFmt.uBitsPerSamp
le==8)?1:2)
                *  m_AudioFmt.ulSamplesPerSec) )
                *  ulNumBytes) );

}

/*****************************************************************
******
 *  Method:
 *              CPNAudioStream::CalcDeviceMs
 *      Purpose:
 *              Calculate the duration in millisecs for this number
 of
 *              bytes in Device format.
 */
```

```
ULONG32 CPNAudioStream::CalcDeviceMs
(
    ULONG32     ulNumBytes
)
{
    return ( (ULONG32) (( 1000.0
                / (m_DeviceFmt.uChannels * ((m_DeviceFmt.uBitsPerSa
mple==8)?1:2)
                *  m_DeviceFmt.ulSamplesPerSec) )
                *  ulNumBytes) );
}

/*****************************************************************
******
 *  Method:
 *              CPNAudioStream::CalcOffset
 *      Purpose:
 *              Calculate the offset in bytes given time.
 */
ULONG32 CPNAudioStream::CalcOffset
(
    ULONG32 ulStartTime
,   ULONG32 ulEndTime
)
{
    /* Using m_ulBytesPerMs may introduce cumulative error due
     * to decimal cutoff
     */
    return (ULONG32) ((ulEndTime - ulStartTime) *
                      (m_ulGranularity ? m_ulInputBytesPerGran*1./m
_ulGranularity : 0));
}


void CPNAudioStream::FlushBuffers(BOOL bInstantaneousAlso)
{
    if ( m_pDataList )
    {
        PNAudioInfo* pInfo = 0;
        CPNSimpleList::Iterator lIter = m_pDataList->Begin();
        for (; lIter != m_pDataList->End(); ++lIter)
        {
            pInfo  = (PNAudioInfo*) (*lIter);
            if ( pInfo )
            {
                pInfo->pBuffer->Release();
                delete pInfo;
                pInfo  = 0;
```

```
            }
        }
        m_pDataList->RemoveAll();
    }

    if ( m_pInstantaneousList && bInstantaneousAlso)
    {
        PNAudioInfo* pInfo = 0;
        CPNSimpleList::Iterator lIter = m_pInstantaneousList->Begin
();
        for (; lIter != m_pInstantaneousList->End(); ++lIter)
        {
            pInfo  = (PNAudioInfo*) (*lIter);
            if ( pInfo )
            {
                pInfo->pBuffer->Release();
                delete pInfo;
                pInfo  = 0;
            }
        }
        m_pInstantaneousList->RemoveAll();
    }
}


BOOL
CPNAudioStream::EnoughDataAvailable(ULONG32& ulLastWriteTime, ULONG
32& ulNumMsRequired)
{
    ULONG32 ulBytesNeeded = 0;
    BOOL    bAvailable = TRUE;
        // ////////////////////////////////////////////////////////
/////
    // If no  resampling, mix stream data directly into the player
    // buffer.
    if ( !m_pResampleId )
    {
        ulBytesNeeded   = m_ulInputBytesPerGran;
    }
    // ////////////////////////////////////////////////////////
/
    // If resampling, mix stream data into a tmp buffer. Then
    // resample this buffer and mix the final resampled buffer into
    // the player buffer.
    else
    {
        /*
         * Audio Session will always ask for m_ulOutputBytesPerGran
```

```
 bytes to be mixed
         * in MixIntoBuffer() call. So we need to produce these man
y number of bytes.
         * If there is any mono-stereo conversion that happens in t
he mixing, number
         * of output bytes required from the resampler are half the
 number of
         * m_ulOutputBytesPerGran bytes.
         */

        ULONG32 ulMaxFramesOut = m_ulOutputBytesPerGran/(m_DeviceFm
t.uBitsPerSample==8 ? 1 : 2);

        if (m_AudioFmt.uChannels == 2 || m_DeviceFmt.uChannels == 2
)
        {
            ulMaxFramesOut /= 2;
        }


        ULONG32 ulMaxFramesIn = ResamplerRequires(ulMaxFramesOut, m
_pResampleId);

        ulBytesNeeded = ulMaxFramesIn *  ((m_AudioFmt.uBitsPerSampl
e==8)? 1 : 2)
                                        * m_AudioFmt.uChannels;
    }

    ULONG32         ulBytesAvailable    = 0;
    LISTPOSITION    lp                  = m_pDataList->GetHeadPosit
ion();

    while(lp)
    {
        PNAudioInfo* pInfo  = (PNAudioInfo*) m_pDataList->GetNext(l
p);

        ulBytesAvailable += pInfo->ulBytesLeft;
        if (ulBytesAvailable >= ulBytesNeeded)
        {
            return TRUE;
        }
    }

    ulNumMsRequired = CalcMs(ulBytesNeeded - ulBytesAvailable);

    PN_ASSERT(m_ulLastWriteTime + m_ulGranularity >= ulNumMsRequire
d);
```

```
    if (m_ulLastWriteTime + m_ulGranularity >= ulNumMsRequired)
    {
        ulLastWriteTime = m_ulLastWriteTime + m_ulGranularity - ulN
umMsRequired;
    }
    else
    {
        ulLastWriteTime = 0;
    }

    return FALSE;
}


void
CPNAudioStream::SetLive(BOOL bIsLive)
{
    if (m_bIsFirstPacket)
    {
        m_bIsLive = bIsLive;
    }
}

PN_RESULT
CPNAudioStream::StartCrossFade(CPNAudioStream*  pFromStream,
                               UINT32           ulCrossFadeStartTim
e,
                               UINT32           ulCrossFadeDuration
,
                               BOOL             bToStream)
{
    if (m_bCrossFadingToBeDone)
    {
        return PNR_UNEXPECTED;
    }

    PN_RELEASE(m_pCrossFadeStream);

    m_bCrossFadingToBeDone  = TRUE;
    m_pCrossFadeStream      = pFromStream;
    m_pCrossFadeStream->AddRef();
    m_ulCrossFadeStartTime  = ulCrossFadeStartTime;
    m_ulCrossFadeDuration   = ulCrossFadeDuration;
    m_bFadeToThisStream     = bToStream;

    if (m_bInited && m_bFadeToThisStream)
    {
```

```
        InitializeCrossFader();
    }

    return PNR_OK;
}

void
CPNAudioStream::InitializeCrossFader(void)
{
    if (!m_pCrossFader)
    {
        m_pCrossFader = new CrossFader;
    }

    UINT16 uNumSamplesToFade = (UINT16)
        (m_DeviceFmt.ulSamplesPerSec * m_ulCrossFadeDuration/1000);

    /* Make cross-fade duration to land on a sample boundary */
    m_ulCrossFadeDuration = (uNumSamplesToFade * 1000)/
                                m_DeviceFmt.ulSamplesPerSec;

    PN_ASSERT(m_ulCrossFadeDuration > 0 && uNumSamplesToFade > 0);

    m_pCrossFader->Initialize(uNumSamplesToFade, m_DeviceFmt.uChann
els);
}
```

Appendix C

---
revision 1.94
date: 1998/ 18:57:22; author: rahul; state: Exp; lines: +73 -2
branches: 1.94.2;
Fixed occasioncal static during cross-fade.
We were not chopping off the data from the To stream at sample boundary.

Fixed wierd sample rate logic to match the closest supported sample rate.

---
revision 1.86
date: 1998/; author: rahul; state: Exp; lines: +4 -2
Bug fix for cross-fade.
---
revision 1.85
date: 1998/; author: henry; state: Exp; lines: +26 -8
More fixes for audio cross-fade (Rahul)
---
revision 1.84
date: 1998/; author: rahul; state: Exp; lines: +252 -93
CrossFade is now functional!
---
revision 1.80
date: 1998/; author: rahul; state: Exp; lines: +42 -20
Code complete on Cross-Fader. Need to be tested once RealAudio renderer
starts using it.
---
revision 1.77
date: 1998/ 07:47:07; author: rahul; state: Exp; lines: +194 -9
Plumming for Cross-Fade support complete.
Need to change one function to add an efficient cross-fader.
---
revision 1.76
date: 1998/ 05:53:23; author: rahul; state: Exp; lines: +35 -45
More setup for cross fade.
